Brian Yu
brian@cs.harvard.edu

David J. Malan
malan@harvard.edu

Parser

The latest version of Python you should use in this course is Python 3.12.

Write an AI to parse sentences and extract noun phrases.

```
$ python parser.py
Sentence: Holmes sat.
          S
  ________|_____
 NP             VP
 |              |
 N              V
 |              |
holmes         sat

Noun Phrase Chunks
holmes
```

When to Do It

By Wednesday, December 31, 2025, 11:59 PM EST

How to Get Help

1. Ask questions via Ed!
2. Ask questions via any of CS50's communities!

Background

A common task in natural language processing is parsing, the process of determining the structure of a sentence. This is useful for a number of reasons: knowing the structure of a sentence can help a computer to better understand the meaning of the sentence, and it can also help the computer extract information out of a sentence. In particular, it's often useful to extract noun phrases out of a sentence to get an understanding for what the sentence is about.

In this problem, we'll use the context-free grammar formalism to parse English sentences to determine their structure. Recall that in a context-free grammar, we repeatedly apply rewriting rules to transform symbols into other symbols. The objective is to start with a nonterminal symbol `S` (representing a sentence) and repeatedly apply context-free grammar rules until we generate a complete sentence of terminal symbols (i.e., words). The rule `S -> N V` , for example, means that the `S` symbol can be rewritten as `N V` (a noun followed by a verb). If we also have the rule `N -> "Holmes"` and the rule `V -> "sat"` , we can generate the complete sentence `"Holmes sat."` .

Of course, noun phrases might not always be as simple as a single word like `"Holmes"` . We might have noun phrases like `"my companion"` or `"a country walk"` or `"the day before Thursday"` , which require more complex rules to account for. To account for the phrase `"my companion"` , for example, we might imagine a rule like:

```
NP -> N | Det N
```

In this rule, we say that an `NP` (a "noun phrase") could be either just a noun (`N`) or a determiner (`Det`) followed by a noun, where determiners include words like `"a"` , `"the"` , and `"my"` . The vertical bar (`|`) just indicates that there are multiple possible ways to rewrite an `NP` , with each possible rewrite separated by a bar.

To incorporate this rule into how we parse a sentence (`S`), we'll also need to modify our `S -> N V` rule to allow for noun phrases (`NP`s) as the subject of our sentence. See how? And to account for more complex types of noun phrases, we may need to modify our grammar even further.

Getting Started

- Download the distribution code from https://cdn.cs50.net/ai/2023/x/projects/6/parser.zip and unzip it.
- Inside of the `parser` directory, run `pip3 install -r requirements.txt` to install this project's dependency: `nltk` for natural language processing.

Understanding

First, look at the text files in the `sentences` directory. Each file contains an English sentence. Your goal in this problem is to write a parser that is able to parse all of these sentences.

Take a look now at `parser.py` , and notice the context free grammar rules defined at the top of the file. We've already defined for you a set of rules for generating terminal symbols (in the global variable `TERMINALS`). Notice that `Adj` is a nonterminal symbol that generates adjectives, `Adv` generates adverbs, `Conj` generates conjunctions, `Det` generates determiners, `N` generates nouns (spread across multiple lines for readability), `P` generates prepositions, and `V` generates verbs.

Next is the definition of `NONTERMINALS` , which will contain all of the context-free grammar rules for generating nonterminal symbols. Right now, there's just a single rule: `S -> N V` . With just that rule, we can generate sentences like `"Holmes arrived."` or `"He chuckled."` , but not sentences more complex than that. Editing the `NONTERMINALS` rules so that all of the sentences can be parsed will be up to you!

Next, take a look at the `main` function. It first accepts a sentence as input, either from a file or via user input. The sentence is preprocessed (via the `preprocess` function) and then parsed according to the context-free grammar defined by the file. The resulting trees are printed out, and all of the "noun phrase chunks" (defined in the Specification) are printed as well (via the `np_chunk` function).

In addition to writing context-free grammar rules for parsing these sentences, the `preprocess` and `np_chunk` functions are left up to you!

Specification

Complete the implementation of `preprocess` and `np_chunk` , and complete the context-free grammar rules defined in `NONTERMINALS` .

- The `preprocess` function should accept a `sentence` as input and return a lowercased list of its words.
  - You may assume that `sentence` will be a string.
  - You should use `nltk`'s `word_tokenize` function to perform tokenization.
  - Your function should return a list of words, where each word is a lowercased string.
  - Any word that doesn't contain at least one alphabetic character (e.g. `.` or `28`) should be excluded from the returned list.
- The `NONTERMINALS` global variable should be replaced with a set of context-free grammar rules that, when combined with the rules in `TERMINALS` , allow the parsing of all sentences in the `sentences/` directory.
  - Each rules must be on its own line. Each rule must include the `->` characters to denote which symbol is being replaced, and may optionally include `|` symbols if there are multiple ways to rewrite a symbol.
  - You do not need to keep the existing rule `S -> N V` in your solution, but your first rule must begin with `S ->` since `S` (representing a sentence) is the starting symbol.
  - You may add as many nonterminal symbols as you would like.
  - Use the nonterminal symbol `NP` to represent a "noun phrase", such as the subject of a sentence.
- The `np_chunk` function should accept a `tree` representing the syntax of a sentence, and return a list of all of the noun phrase chunks in that sentence.
  - For this problem, a "noun phrase chunk" is defined as a noun phrase that doesn't contain other noun phrases within it. Put more formally, a noun phrase chunk is a subtree of the original tree whose label is `NP` and that does not itself contain other noun phrases as subtrees.
    - For example, if `"the home"` is a noun phrase chunk, then `"the armchair in the home"` is not a noun phrase chunk, because the latter contains the former as a subtree.
  - You may assume that the input will be a `nltk.tree` object whose label is `S` (that is to say, the input will be a tree representing a sentence).
  - Your function should return a list of `nltk.tree` objects, where each element has the label `NP` .
  - You will likely find the documentation for `nltk.tree` helpful for identifying how to manipulate a `nltk.tree` object.

You should not modify anything else in `parser.py` other than the functions the specification calls for you to implement, though you may write additional functions and/or import other Python standard library modules. You will need to modify the definition of `NONTERMINALS` , but you should not modify the definition of `TERMINALS` .

Hints

- It's to be expected that your parser may generate some sentences that you believe are not syntactically or semantically well-formed. You need not worry, therefore, if your parser allows for parsing meaningless sentences like `"His Thursday chuckled in a paint."`
  - That said, you should avoid over-generation of sentences where possible. For example, your parser should definitely not accept sentences like `"Armchair on the sat Holmes."`
  - You should also avoid under-generation of sentences. A rule like `S -> N V Det Adj Adj Adj N P Det N P Det N` would technically successfully generate sentence 10, but not in a way that is particularly useful or generalizable.
  - The rules in the lecture source code are (intentionally) a very simplified rule set, and as a result may suffer from over-generation. You can (and should) make modifications to those rules to try to be as general as possible without over-generating. In particular, consider how you might get your parser to accept the sentence "Holmes sat in the armchair." (and "Holmes sat in the red armchair." and "Holmes sat in the little red armchair."), but have it *not* accept the sentence "Holmes sat in the the armchair."
- It's to be expected that your parser may generate multiple ways to parse a sentence. English grammar is inherently ambiguous!
- Within the `nltk.tree` documentation, you may find the `label` and `subtrees` functions particularly useful.
- To focus on testing your parser before working on noun phrase chunking, it may be helpful to temporarily have `np_chunk` simply return an empty list `[]` , so that your program can operate without noun phrase chunking while you test the other parts of your program.

Testing

If you'd like, you can execute the below (after setting up `check50` on your system) to evaluate the correctness of your code. This isn't obligatory; you can simply submit following the steps at the end of this specification, and these same tests will run on our server. Either way, be sure to compile and test it yourself as well!

```
check50 ai50/projects/2024/x/parser
```

Execute the below to evaluate the style of your code using `style50` .

```
style50 parser.py
```

Remember that **you may not import any modules** (other than those in the Python standard library) **other than those explicitly authorized herein**. Doing so will not only prevent `check50` from running, but will also prevent `submit50` from scoring your assignment, since it uses `check50` . If that happens, you've likely imported something disallowed or otherwise modified the distribution code in an unauthorized manner, per the specification. There are certainly tools out there that trivialize some of these projects, but that's not the goal here; you're learning things at a lower level. If we don't say here that you can use them, you can't use them.

How to Submit

Beginning **Monday, January 1, 2024, 12:00 AM EST**, the course has transitioned to a new submission platform. If you had not completed CS50 AI prior to that time, **you must join the new course pursuant to Step 1, below**, and also must resubmit all of your past projects using the new submission slugs to import their scores. We apologize for the inconvenience, but hope you feel that access to `check50` , which is new for 2024, is a worthwhile trade-off for it, here!

1. Visit this link, log in with your GitHub account, and click **Authorize cs50**. Then, check the box indicating that you'd like to grant course staff access to your submissions, and click **Join course**.
2. Install Git and, optionally, install `submit50` .
3. If you've installed `submit50` , execute

   ```
   submit50 ai50/projects/2024/x/parser
   ```

   Otherwise, using Git, push your work to `https://github.com/me50/USERNAME.git` , where `USERNAME` is your GitHub username, on a branch called `ai50/projects/2024/x/parser` .

If you submit your code directly using Git, rather than `submit50` , **do not** include files or folders other than those you are actually instructed to modify in the specification above. (That is to say, don't upload your entire directory!)

Work should be graded within five minutes. You can then go to https://cs50.me/cs50ai to view your current progress!